How well are regular expressions tested in the wild?

Peipei Wang
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
pwang7@ncsu.edu

Kathryn T. Stolee
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
ktstolee@ncsu.edu

ABSTRACT

Developers report testing their regular expressions less than the rest of their code. In this work, we explore how thoroughly tested regular expressions are by examining open source projects.

Using standard metrics of coverage, such as line and branch coverage, gives an incomplete picture of the test coverage of regular expressions. We adopt graph-based coverage metrics for the DFA representation of regular expressions, providing fine-grained test coverage metrics. Using over 15,000 tested regular expressions in 1,225 Java projects on GitHub, we measure node, edge, and edge-pair coverage. Our results show that only 17% of the regular expressions in the repositories are tested at all. For those that are tested, the median number of test inputs is two. For nearly 42% of the tested regular expressions, only one test input is used. Average node and edge coverage levels on the DFAs for tested regular expressions are 59% and 29%, respectively. Due to the lack of testing of regular expressions, we explore whether a string generation tool for regular expressions, Rex, achieves high coverage levels. With some exceptions, we found that tools such as Rex can be used to write test inputs with similar coverage to the developer tests.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Language features*;

KEYWORDS

Regular expressions, Test coverage metrics, Deterministic Finite Automaton

ACM Reference Format:
Peipei Wang and Kathryn T. Stolee. 2018. How well are regular expressions tested in the wild?. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236072

1 INTRODUCTION

A survey of professional developers reveals that they test their regular expressions *less* than the rest of their code [9]. In this work, we explore how thoroughly tested regular expressions are by examining open source projects.

Traditional code coverage criteria, are rather coarse-grained when it comes to regular expressions. Statement coverage requires the regular expression to be invoked at least once. If the regular expression call site appears in a predicate, branch coverage requires that the regular expression is tested with at minimum two strings, one in the language of the regular expression and one not. However, these metrics ignore the complex structure represented by a regular expression. We propose to use test metrics for graph-based coverage [2] over the DFA representation of regular expressions.

Regular expression tools can help support developers in their creation and testing of regular expressions. These tools either automatically generate strings according to the given regular expressions [20, 21, 28, 33] or automatically generate regular expressions according to the given list of strings [5, 26]. Rex [33] is a tool for analyzing regular expressions through symbolic analysis. Given a regular expression R, Rex uses the Z3 [15] SMT solver to generate members of the language by treating it as a satisfiability problem. Like automatic test case generation tools, integrating these generated results into software testing can help automate the process, but it is not clear how well covered the regular expressions would be compared to developer-written tests.

In this work, we focus on empirically measuring how well tested regular expressions are and further explore the potential for using existing tools, specifically Rex, to improve the test coverage. First, we measure the test coverage of regular expressions in the wild based on a set of 1,225 Java projects on GitHub containing 15,096 tested regular expressions. Second, we measure the test coverage of strings generated by Rex and compare the coverage achieved against the strings generated by developers in the GitHub projects. Our contributions are:

- Application of graph-based metrics for test coverage of regular expressions: node coverage, edge coverage, and edge-pair coverage (Section 3).
- Test coverage evaluation of 15,096 regular expressions based on nearly 900,000 input strings from 1,225 Java projects from GitHub (RQ1).
- Evaluation of test coverage achieved by the Rex symbolic analysis tool for regular expressions (RQ2).

Our main findings are:

- Of 18,426 call sites for three pattern matching API methods identified statically in 1,225 GitHub projects, only 3,093 (16.8%) are ever executed by test suites (RQ1).
- Of 15,096 regular expressions captured during test suite execution of 1,225 GitHub projects, 10,970 (72.7%) use only failing inputs (4,941) or only matching inputs (6,029) (RQ1).

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236072

- The Rex-generated test inputs achieve similar coverage levels to the developer-written tests (RQ2).

2 BACKGROUND AND MOTIVATION

A regular expression is a sequence of characters that defines a search pattern. The set of strings matched by the regular expression is a language. That is, a regular expression R represents a language $L(R)$ over an alphabet Σ, where $L(R)$ is a (possibly infinite) set of strings. For a given language, there are many regular expressions that can describe it. A regular expression can be represented as a string of tokens, a finite state automaton in deterministic (DFA) form, or in non-deterministic (NFA) form.

In this work, we explore test coverage metrics over the DFA representing a regular expression. This requires three informal explorations to ensure feasibility and assess the potential impact. First, we explore the potential of building DFAs from regular expressions by analyzing regular expressions collected from an existing Python dataset [9] and testing them for regularity [32]. Second, we show intuitively how existing coverage metrics are insufficient. Third, to motivate the structural coverage metrics, we explore whether faults can lie along untested paths in a DFA.

2.1 How Regular are Regular Expressions?

Regular expressions in source code can contain non-regular features, such as backreferences. An example is the regular expression `([a-z]+\1)`, which matches a repeated word in a string, such as "appleapple". Building a DFA is not possible for this since this regular expression is non-regular. For regular expressions in source code that are indeed regular, we can build DFAs and measure coverage based on a test suite. Here, we are testing how many of the regular expressions in the wild are truly regular.

We explore an existing and publicly available dataset of 13,597 regular expressions scraped from Python projects on GitHub. To test for regularity, we use an empirical approach since the ability to build a DFA from a regular expression implies that it is regular [32]. Of the 13,597 Python regular expressions, 13,029 (95.9%) are regular in that we were successful in building DFAs for each using the RE2 [14] regular expression processing engine. For the remaining 568, we investigated each by hand. One regular expression was removed because its repetition exceeds the RE2 limits. While it may indeed be regular, to be conservative, we mark it as non-regular. An additional 81 contained comments within the regular expressions, which are unsupported in RE2, so these were also assumed to be non-regular; 128 contained unsupported characters. The remaining 368 were non-regular as they contained backreferences.

In the end, with nearly 96% of the regular expressions being regular (as a low estimate), we conclude that most regular expressions found in the wild are regular and thus can be modeled with DFAs.

2.2 Limitations of Code Coverage

In this work, we posit that code coverage metrics [2, Chapter 2] [24, 31, 37] such as statement, branch, and path, are too coarse-grained for regular expressions. Statement coverage requires that the code containing the regular expression is reached, leading to a minimum of one test input for the regular expression. If the regular expression is in a statement where the control flow is dependent on the matching outcome, branch coverage requires that the regular expression have at least two inputs, one that evaluates to true and another that evaluates to false.

Consider the following Java code snippet. The *call site* for method `Pattern.matches` is on line 1. The regular expression is `-d|--data`.

```
1  if(Pattern.matches("-d|--data",strInput)){
2      System.out.println("YES");
3      ...
4  }else{
5      System.out.println("NO");
6      ...
7  }
```

Statement coverage of the regular expression requires that line 1 is executed and branch coverage requires two test inputs, one to cover the `true` branch and one to cover the `false` branch. Using coverage metrics based on the DFA representation of the regular expression, on the other hand, would require 1) each branch to be covered, and 2) each *case* in the regular expression, "-d" and "`--data`", to be covered. Such metrics measure test coverage of the regular expression's control flow (i.e., the DFA) just like branch coverage measures test coverage of source code's control flow graph.

Existing tools and techniques can direct test input generation toward areas of untested paths. One technique among these is symbolic execution [3, 8, 18, 22, 25], and Rex [33] has been developed for symbolic analysis of regular expressions. However, Rex focuses solely on the matching behavior [33], which limits its ability to cover the false branch in the Java example above. Hampi [20, 21] and brics [28] similarly only generates passing strings. While useful, there are no guarantees of structural coverage.

2.3 DFA Coverage Example

Bug reports related to regular expressions abound. A search for "regex OR regular expression" in GitHub yields over 555,000 issues, with 22% of those still being open. One in particular illustrates how coverage metrics on the DFA could have brought a particular bug to the developer's attention sooner. This bug report[1] describes an issue with the regular expression `\d+\.d+` in the NAR plugin for Maven. Figure 1 shows the DFA of this regular expression built using RE2 [14], and we take this opportunity to describe the DFA notation used throughout this paper[2].

Node 0 is the start-state, indicated by the incoming arrow. Nodes with double-circles are accept states, such as Node 4. Node e is the error state, denoting a mismatch. The edges are labeled with transitions, often using syntactic sugar for ease of interpretation. The edge $\overrightarrow{01}$ is traversed when a digit from `0-9` is read. If any other character is read at Node 0, (i.e., `not 0-9`), edge $\overrightarrow{0e}$ is traversed. There is a self-loop on Node 1 for digits `0-9`. If the period character is read from Node 1, then edge $\overrightarrow{12}$ is traversed.

In RE2, when reading an input string, byte `[256]`, is added as a text-end marker. For example, the input string "0.0" is transformed to the byte stream `[48 46 48 256]`, as `[48]` is the byte for '0', `[46]` is for '.', and `[256]` marks the end of the string. Byte [256] is matched on edges '`[0-256]`', '`not 0-9`', '`not d`', or '`any except 0-9 and .`'.

[1] https://github.com/maven-nar/nar-maven-plugin/issues/228

[2] The regular expression in the bug is triggered by `Matcher.find()` with a ManyMatch DFA. For simplicity, we show the FullMatch DFA, a subgraph of the ManyMatch.

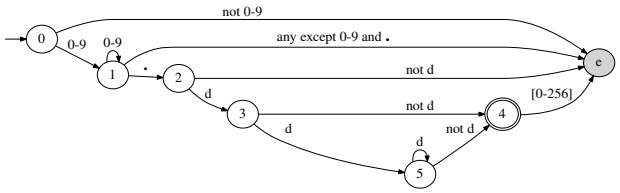


Figure 1: Full-match DFA for regular expression: `\d+\.d+`

The bug report mentions that the regular expression `\d+\.d+` is buggy and the patch adds an escape before the second d, `\d+\.\d+`. The intended behavior is to match input strings with one or more digits, followed by a period, followed by one or more digits.

In this work, the structural metrics could reveal this fault. With the DFA in Figure 1, when Node 3 is reached, the fault may be revealed. Input "0.d" traverses $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ and ends in an accept state, when it should fail. However, input "0.d3" traverses $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow e$ and ends in an error state, as expected. Covering edge $\overrightarrow{2e}$ may also reveal the fault; input "2.3" traverses $0 \rightarrow 1 \rightarrow 2 \rightarrow e$ and ends in and error state, when it should be accepted. Requiring coverage of all feasible nodes and edges could have revealed this fault in the regular expression.

As with code coverage, uncovered artifacts alert the programmer to untested behavior. Such coverage information can indicate that a regular expression is not well tested and for some inputs it may not behave as intended, as is the case here.

3 TEST COVERAGE METRICS

We explore fine-grained coverage metrics for regular expressions based on a DFA representation. The intuition is that since regular expressions are equivalent to DFAs [32], and 96% of regular expressions in the wild were found to be regular (Section 2.1), then graph coverage metrics over the DFA can be used to test the behavior within most regular expressions. We discuss three levels of coverage: Node Coverage (NC), Edge Coverage (EC), and Edge-Pair Coverage (EPC). These coverage metrics are adopted from graph coverage metrics proposed by Ammann and Offutt [2, Chapter 7].

3.1 Graph Notation

For ease of exposition, we expand on the traditional definition of a DFA. In this work, a DFA graph $G = \{N, N_0, N_m, N_e, E\}$ where: N is the set of all nodes, N_0 is the initial node, N_m is the final matching/accept node, N_e is the final failing/error node, and E is a set of all edges. For the DFAs in this work, there is only one initial state, one accept state, and one error state.[1]

The states in a DFA are the nodes $N = \{n_0, n_1, \ldots, n_k\}$. For any two nodes n_1 and n_2 such that $\{n_1, n_2\} \subseteq N$, if there is a transition from n_1 to n_2 in DFA, then the edge $\overrightarrow{n_1 n_2} \in E$; the start and end-state of the path may be the same node, as is the case of self-loops. Edge pairs are defined by paths of length two in the DFA. For example, if $\{\overrightarrow{n_1 n_2}, \overrightarrow{n_2 n_3}\} \in E$, we denote the edge pair as $\overrightarrow{n_1 n_2 n_3}$. In the case of self-loops, $\overrightarrow{n_2 n_2 n_2}$ is also a valid edge-pair.

Given an input string and a regular expression, the initial node N_0 is visited first. Transitions are taken as each character in the string is consumed. The result of the matching process ends in either the accept node N_m or the error node N_e. In standard DFAs, a traversal can end in any node. However, the DFA generation algorithm used in this work is based on the RE2 tool, which always ends processing in an explicit matching/accept (N_m) or error (N_e) state. In this tool, given a regular expression and an input string, the input string is interpreted as a byte stream, with byte `[256]` added to the end to mark the end of the string. Thus, an input string "2" would be interpreted as `[50 256]` and the input string "1001" would be interpreted as `[49 48 48 49 256]`.

[1] In a FullMatch DFA (see Section 5.1), there could be several matching nodes, and only one accept. We simplified to use only one accept state.

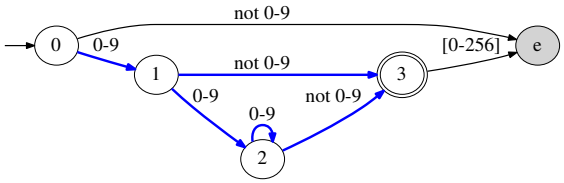


Figure 2: Full-match DFA from RE2 [14] for the regular expression `\d+`. RE2 interprets every string as a byte stream; the range of bytes is `[0-256]` where `[256]` is added to mark the end of a string. Thus, the input string "2" would be represented as `[50 256]` and traverse the following path: $0 \rightarrow 1 \rightarrow 3$. The edges marked *0-9* represent the byte range `[48-57]`; edges *not 0-9* represent the byte ranges `[0-47][58-256]`.

As a running example, consider regular expression R = `\d+` and graph G in Figure 2. In G, $N = \{0, 1, 2, 3, E\}$, $N_0 = 0$, $N_m = 3$, $N_e = e$, $E = \{\overrightarrow{0e}, \overrightarrow{01}, \overrightarrow{12}, \overrightarrow{13}, \overrightarrow{22}, \overrightarrow{23}, \overrightarrow{3e}\}$, and $EP = \{\overrightarrow{012}, \overrightarrow{013}, \overrightarrow{122}, \overrightarrow{123}, \overrightarrow{13e}, \overrightarrow{222}, \overrightarrow{223}, \overrightarrow{23e}\}$. Edges *0-9* cover bytes `[48-57]`, and edges *not 0-9* cover the byte ranges `[0-47][58-256]`; we use the decimal representation to improve clarity.

At this point, we note that this is not the smallest DFA for the regular expression `\d+`. As the same tool is used for the construction of all the DFAs, any impact of the DFAs not being minimal (e.g., extra nodes or edges compared to the minimal representation) is distributed throughout the whole data set and consistent across all experiments. While we refer to RE2 [14] for full details of the DFA construction, though some intuition is provided in Section 5.2.2.

3.2 Coverage Criteria

Given a set of strings S and a DFA G, for all $n \in N$, we mark n as *covered* if n is visited during the processing of some $s \in S$. Similarly, edges $e \in E$ and edge-pairs $ep \in EP$ are marked as *covered* if they are traversed during the processing of some $s \in S$. The sets of covered nodes, edges, and edge-pairs are denoted N_{cov}, E_{cov}, and EP_{cov}, respectively. These sets are aggregated over all $s \in S$.

As defined in prior work [2], we adopt coverage definitions for node coverage (*NC*), edge coverage (*EC*), and edge-pair coverage (*EPC*) as follows:

Definition 3.1 (Node Coverage %). $NC = 100 \times \frac{|\ N_{cov}\ |}{|\ N\ |}$

Definition 3.2 (Edge Coverage %). $EC = 100 \times \frac{|\ E_{cov}\ |}{|\ E\ |}$

Definition 3.3 (Edge-Pair Coverage %). $EPC = 100 \times \frac{|\ EP_{cov}\ |}{|\ EP\ |}$

To illustrate the coverage levels, consider the graph G for the regular expression `\d+` in Figure 2 and the string s_0 ="2" with

Table 1: Coverage of \d+: S = {"2", "1001", "u", "100u"}, S_{succ} = {"2", "1001"}, **and** S_{fail} = {"u", "100u"}.

	S	S_{succ}	S_{fail}
NC	100.0%	80.0%	100.0%
EC	100.0%	71.4%	85.7%
EPC	75.0%	62.5%	50.0%

$S = \{s_0\}$. Traversing G visits $0 \rightarrow 1 \rightarrow 3$ (recall that "2" is interpreted as the byte stream [50 256]). Node 3 is the accept node, which denotes that the regular expression matches the input string (i.e., $s \in L(R)$). During the traversal of G, nodes {0, 1, 3} are visited, meaning that $N_{cov} = \{0, 1, 3\}$, $E_{cov} = \{\overrightarrow{01}, \overrightarrow{13}\}$, and $EP_{cov} = \{\overrightarrow{013}\}$. The coverage levels for \d+ by input strings $S = \{s_0\}$ are: NC = 60% (3/5), EC = 28.6% (2/7), and EPC = 12.5% (1/8).

Next, consider adding the string s_1 ="1001", which is interpreted as the byte stream `[49 48 48 49 256]`. Now, $S = \{s_0, s_1\}$. Traversing G on s_1 traverses the following path: $0 \rightarrow 1 \rightarrow 2 \rightarrow 2 \rightarrow 2 \rightarrow 3$, adding node 2 to N_{cov}, edges $\overrightarrow{12}$, $\overrightarrow{22}$, and $\overrightarrow{23}$ to E_{cov}, and edge-pairs $\overrightarrow{012}$, $\overrightarrow{122}$, $\overrightarrow{222}$, and $\overrightarrow{223}$ to EP_{cov}. As a result, the coverage levels for the regular expression \d+ by input strings $S = \{s_0, s_1\}$ are: NC = 80% (4/5), EC = 71.4% (5/7), and EP = 62.5% (5/8).

As an example of a non-matching string, let s_2 = "u", which is interpreted as the byte stream `[117 256]`. The path traversed in G is $0 \rightarrow e$; after reaching e, the processing stops. Node e is added to N_{cov}, edge $\overrightarrow{0e}$ is added to E_{cov}, and there is no change to EP_{cov}. Considering $S = \{s_0, s_1, s_2\}$, the combined coverage levels are: NC = 100% (5/5), EC = 85.7% (6/7), and EPC = 62.5% (5/8).

For another example of a non-matching string, let s_3 = "100u", which is interpreted as the byte stream `[49 48 48 117 256]`. The path traversed in G is $0 \rightarrow 1 \rightarrow 2 \rightarrow 2 \rightarrow 3 \rightarrow e$. While this input visits all nodes in G, NC = 100% already, so no nodes are added to N_{cov}. Edge $\overrightarrow{3e}$ is added to E_{cov}, edge-pair $\overrightarrow{23e}$ is added to EP_{cov}. Considering $S = \{s_0, s_1, s_2, s_3\}$, the combined coverage levels are: NC = 100% (5/5), EC = 100% (7/7), and EPC = 75% (6/8).

For each coverage metric, we compute coverage over the entire set of input strings, *total*, and two subsets: *success*, and *failure*. The numbers reported in this section are for the *total* set of input strings, that is, $S = \{s_0, s_1, s_2, s_3\}$. After, we split the input strings into those that terminate in an accept state in N_m, which we call S_{succ}, and those that terminate in the error state N_e, which we call S_{fail}. With this example, $S_{succ} = \{s_0, s_1\}$ and $S_{fail} = \{s_2, s_3\}$.

Table 1 presents a summary of the coverage levels for each set of input strings. Achieving 100% for any of the coverage metrics is infeasible for S_{succ} alone because the error state e will never be reached, missing that node and the edges leading to it. In this example, EC for S_{succ} is 71.4% while EC for S is 100%.

Achieving 100% coverage for EPC is the most difficult, but it is possible in this example. The missing edge-pairs are computed by $EP \setminus EP_{cov} = \{\overrightarrow{123}, \overrightarrow{13e}\}$. Two additional input strings can lead to 100% EPC. Input "1u" would be interpreted as the byte stream `[49 117 256]` and traverses the path $0 \rightarrow 1 \rightarrow 3 \rightarrow e$, hence covering $\overrightarrow{13e}$. Input "11u" would lead to byte stream `[49 49 117 256]`, traverse the path $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow e$ and cover $\overrightarrow{123}$.

Note that it is possible to have a DFA which is simply two nodes connected by a single edge. Thus, edge pairs may not exist. For this case, we treat edge-pair coverage as identical to edge coverage. Among the 15,096 regular expressions studied in this work, only two regular expressions have this structure.

4 RESEARCH QUESTIONS

To explore the potential of using graph coverage metrics for regular expressions, we evaluate the following research questions:

RQ1: *How well are regular expressions tested in GitHub?*

To answer RQ1, we identify 1,225 Java projects that have existing test suites covering the regular expressions. From these, we extract 15,096 regular expressions and 899,804 total test input strings, measuring NC, EC, and EPC for each regular expression. To obtain the regular expressions and their corresponding strings which are covered by test cases, we use the Java bytecode manipulation framework Javassist [11] to record the regular expressions when pattern matching methods are triggered by test cases.

RQ2: *How well can the regular expression string generation tool Rex improve the test coverage of regular expressions?*

Using the regular expressions from RQ1, we generate test strings using Rex [33] and calculate the regular expression coverage, comparing it to the coverage of the user-defined test suites from RQ1. Using Rex, we generate test suites of three sizes, one to match the size of the user-defined test suites from the GitHub projects, one 5x that size, and one 10x that size. By comparing the coverage statistics we got in RQ2 to those in RQ1, we evaluate the test coverage possibilities through using an automated tool.

5 STUDY

Applying the coverage metrics defined in Section 3.2 to regular expressions from the wild requires (1) instrumentation to capture the regular expressions and strings matched against them (Section 5.1), (2) a tool to measure coverage given a regular expression and a set of strings (Section 5.2), and (3) a large corpus of projects with regular expressions and test suites that execute the regular expressions (Section 5.3). To address RQ2, we use the Rex [33] tool to generate input strings for the regular expressions in our study (Section 5.4).

5.1 Instrumentation

This section describes our approach to collecting regular expressions from GitHub projects and the strings evaluated against the regular expressions during testing.

5.1.1 Instrumented Functions. There are different types of matching between a regular expression and a string. The Java function *Pattern.matches* requires the regular expression to match a string from its beginning to its end; Python's *re.match* requires the regular expression to match a string only from its beginning, not necessarily match to the end of the string; and the C# function *Regex.Match* requires the regular expression to match only a substring of the input string. These are called *FullMatch*, *FirstMatch*, and *ManyMatch*, respectively. In this work, we consider only *FullMatch* matches and related functions in Java projects. The related functions for FullMatch in Java are:

- `java.lang.String.matches(String regex)`
- `java.util.regex.Matcher.matches()`

- `java.util.regex.Pattern.matches(String regex, CharSequence input)`

In these functions the entire string is required to match the regular expression [17]. Thus, a regular expression with end-point anchors (i.e., ^ and $) and without are no different.

5.1.2 Bytecode Manipulation. Our instrumentation is built on top of the Java bytecode manipulation framework Javassist [11], which can dynamically change the class bytecode in the JVM. All the projects are run in jdk1.7. We intercepted FullMatch function invocations in Java. For each invocation, we collect information about the regular expression itself, its location in the code, and any strings matched against it during test suite execution. These strings matched against the regular expression are referred to as the *input strings* or *test inputs* (i.e., S from Section 3.2).

Since a regular expression may also appear in third-party libraries, we use the Java Reflection API to additionally record the caller function stack of the instrumented methods and extract the file name, class name, and method name of their caller methods. This allows us to identify when the regular expression being executed is from the system under test and when it is from a third-party library. We are dependent on two libraries during the experimentation, `org.junit` and `org.apache.maven`. Because Maven uses regular expressions to automate unit tests, all recorded regular expressions whose test classes are from package `org.junit.runner.*` or from package `org.apache.maven.plugins.*` are treated as regular expressions from third-party libraries and dropped.

5.1.3 Recorded Information. We illustrate the recorded information for the regular expression `((:\w+)|\*)` and a string "one-name" from a project used in our study[1]:

- system under test: `mikko-apo/KiRouter.java`
- test file: `SinatraRouteParser.java`
- test class: `kirouter.SinatraRouteParser`
- test method: `compileRoutePattern`
- call site: `line 38`
- regular expression: `((:\w+)|\*)`
- input string:"one-name"

In Section 2.2, the regular expression in the *call site* on line 1 is hard-coded. However, often the regular expression is passed as a variable, allowing multiple regular expressions to be observed during testing at the same call site (i.e., there is a many-to-one relationship between regular expressions and call sites). When this occurs, the recorded information is the same as above, except *regular expression* and *input string* would be different.

5.2 Coverage Analysis

This section details the construction of DFAs for computing coverage. Given a regular expression R and a set of input strings S, we first build a DFA for $L(R)$ and then track the nodes and edges visited in the DFA during pattern matching with each string $s \in S$. We built our infrastructure on top of RE2 [14], a regular expression engine similar to those used in PCRE, Perl, and other languages.[2]

5.2.1 DFA Types. Given a regular expression and an input string to match, we could build multiple DFAs with different considerations. We could build a static DFA with a regular expression alone or build a DFA on-the-fly (dynamic DFA) considering both a regular expression and an input string. For the same regular expression, different input strings will yield different dynamic DFAs. We can also build a Forward DFA and Backward DFA depending on the direction of scanning the regular expression. These decisions come with various performance tradeoffs during the matching process. For the purpose of our work, we need each DFA to be built consistently regardless of the input string, so we use a static DFA. We chose the forward direction as it seems the most natural for interpretation.

5.2.2 DFA Mapping. When matching an input string to a regular expression, RE2 builds a dynamic DFA. However, our coverage is computed over a static DFA. This requires mapping to aggregate coverage of a regular expression given multiple input strings.

For a single regular expression, different input strings often result in different dynamic DFAs. To make matters worse, these DFAs have inconsistent naming of their states. Therefore, to calculate the coverage of a certain regular expression based on the same DFA, these dynamic DFAs have to be mapped to the same static DFA, and then coverage is computed on the static DFA. This is usually straightforward as the dynamic DFA is always an isomorphic subgraph of the static DFA and N_0, N_e and N_m are consistently labeled in the static and dynamic DFAs.

Consider the regular expression `\d+` and $S = \{s_0, s_1, s_2, s_3\}$ from Section 3.2 where s_0 = "2", s_1 = "1001", s_2 = "u", and s_3 = "100u". Figure 3a shows the static forward DFA. The dynamic DFAs corresponding to these four inputs are shown in Figure 3b, Figure 3c, Figure 3d, and Figure 3e, respectively. Blue arrows are used to identify the visited edges in the dynamic DFAs when the input string is a match. Red edges are used to identify the visited edges when the input string is not a match. Note that in Figure 3, for simplicity, we have already mapped and renamed the nodes in the dynamic DFAs according to the static DFA.

5.2.3 RE2 Limitations and Modifications. We enlarged the default memory size of a cached DFA so that it could accommodate large DFA graphs. Due to Linux environment limitations, string length is limited to 131,072 and null type is not allowed. These situations are rare, impacting < 1% of the collected regular expressions (see Section 5.3).

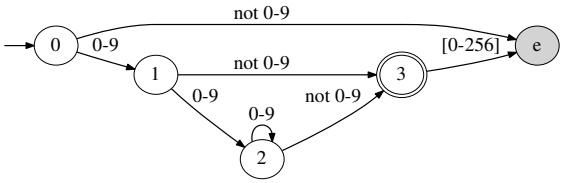
5.2.4 Coverage Calculation. With the consistent naming between a static DFA and a dynamic DFA, all nodes, edges, and edge pairs in the latter are regarded as visited nodes, edges, and edge pairs of the former. That is, a node only appears in a dynamic DFA when it is visited during matching; these can be thought of as *just-in-time* DFA constructions in the context of a string to match. The coverage metrics from Section 3.2 are computed over the static DFAs, aggregating over all input strings observed during testing.

5.3 Artifacts for RQ1

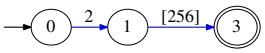
RepoReaper [29] provides a curated list of GitHub projects with the ability to sort based on project properties, such as the availability of test suites, which is a pre-requisite for our study. We focused on

[1] https://github.com/mikko-apo/KiRouter.java

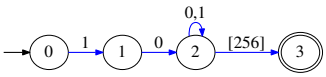
[2] Original RE2 at https://github.com/google/re2 and modified code at https://github.com/wangpeipei90/re2



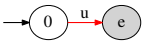
(a) Fully specified static DFA for: `\d+`



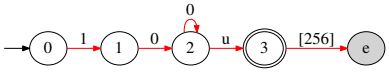
(b) Dynamic DFA for regular expression: `\d+` **and input: "2"**



(c) Dynamic DFA for regular expression: `\d+` **and input: "1001"**



(d) Dynamic DFA for regular expression: `\d+` **and input: "u"**



(e) Dynamic DFA for regular expression: `\d+` **and input: "100u"**

Figure 3: Visited DFA subgraphs for the regular expression '`\d+`'. For each figure, N_0 is the initial node 0, N_m is the accept node 3, N_e is the error node e. The arrows colored blue represent transitions in successful matches. The arrows colored red represent transitions in failed matches. The characters without square brackets are the literal characters in state transitions. For example, 'u' prompts the transition from Node 0 to Node e. `[256]` implies that there are no more bytes from the input string.

Java projects due to its popularity on GitHub and the availability of a bytecode analysis framework for instrumentation.

5.3.1 Project Selection. In December 2017, we selected the 136,196 Java projects whose unit test ratio reported in RepoReaper is greater than zero. Because the density of regular expressions in projects tends to be low, we automated project builds and test suite execution in order to collect sufficient data. As such, we require all projects we analyze to use *maven* and *junit* to automatically run unit tests. We identified 13,637 Java Maven projects that used Java pattern matchings functions mentioned in Section 5.1. From those, we selected the ones that could be successfully compiled and tested in Maven, leaving 5,691 projects on which we attempted to collect coverage information.

5.3.2 Regular Expression and Test Input Collection. To collect the input strings for each regular expression, we instrumented each project and executed the test suites. We changed the configurations of the plugin *maven-surefire-plugin* by adding *-javaagent* argument to *argLine* so that when Maven forks a VM to run the unit tests the VM can load the instrumentation library. Each project module that runs tests executes in different VMs and the information is recorded in different files. *testFailureIgnore* is configured to *true* so that one test failure does not affect the other tests, allowing us to record as many regular expressions in the project as possible.

Table 2: Description of 1,225 Java Projects Analyzed. All numbers are rounded to nearest integer except the test ratio and KLOC.

Attributes	mean	25%	50%	75%	90%	99%
Tested Regular exp.	12	1	3	7	18	99
Stars	35	0	1	5	30	833
Test ratio	0.238	0.096	0.210	0.346	0.482	0.691
KLOC	55.4	2.0	6.7	25.1	86.7	951.0
Size (KB)	19,062	286	1,079	6,449	33,163	249,915
Call sites	15	2	4	10	31	211
Tested call sites	3	1	2	3	6	20
Reg. exp./tested site	5	1	1	2	5	38

Of the 5,691 projects with Maven, test suites, and pattern matching functions, 1,665 projects contained 24,058 regular expressions executed by test suites. The remaining projects contained regular expressions *not* executed by the test suites, and thus could not be instrumented.

5.3.3 Filtering Out Third-Party Regular Expressions. FullMatch invocations from Maven and JUnit have been removed already at this point, but other third-party libraries also use regular expressions. We can detect this by looking for syntactically identical regular expressions with invocations on the same file, same class, same method, but in different GitHub projects. If the number of projects is larger than one, then it is regarded as a third-party regular expression, and all records related to the same stack information are dropped. A limitation of this approach is that we miss some third-party invocations that are only present in a single project. Given the large number of projects analyzed, the impact of this is likely to be small.

We identified 8,496 regular expressions as coming from third-party libraries. The resulting dataset contains 1,256 projects and 15,562 regular expressions, 14,040 of which are syntactically unique.

5.3.4 RE2 Analysis. Since RE2 only supports the most common regular expression language features, we filtered out the regular expressions containing advanced and non-regular features. RE2 failed to construct DFAs for 457 regular expressions, leaving 15,105 regular expressions spread across 1,225 projects.[1] The RE2 limitations on input string length and the null byte affected 56 regular expressions and 191 input strings, and nine of the 56 regular expressions are removed from coverage analysis because their only input string is dropped.

These 1,225 projects contain 18,426 call sites of the instrumented functions. Only 3,093 call sites are executed by the test suites; the same call site can have many regular expressions in the case of dynamically generated regular expressions.

The final dataset used for analysis contains 1,225 projects, 3,093 call sites, 15,096 regular expressions, of which 13,632 are syntactically unique. As the same regular expression can appear in multiple projects, or multiple places in the same project, all are retained since each is potentially tested differently. These 15,096 regular expressions are executed by 899,804 test inputs.

[1] Assuming all 457 are non-regular, this means over 97% of the regular expressions sampled are regular, echoing findings from the Python analysis in Section 2.1.

Table 3: Description of 15,096 regular expressions analyzed for RQ1. All numbers are rounded to nearest integer.

Attributes	mean	25%	50%	75%	90%	99%
Nodes ($\|N\|$)	144	12	28	70	324	939
Edges ($\|E\|$)	565	24	75	212	938	2,813
Edge pairs ($\|EP\|$)	2,115	25	99	414	1,647	16,850
Regular exp. len.	31	13	18	39	67	161
# Input strings ($\|S\|$)	60	1	2	7	27	662
Input string len.	125	9	17	63	318	948

5.3.5 Project Characteristics. Table 2 describes the 1,225 projects in terms of *Tested regular exp.* (numbers of tested regular expressions per project), *Stars* (a measure of popularity), *KLOC* (lines of code in thousands), *Size* (size of the repository in KB), *Test ratio* (the ratio of number of lines of code in test files to the total lines of code in repository, as reported by RepoReaper), *Call sites* (the number of FullMatch methods in the source code), *Tested call sites* (the number of FullMatch call sites executed by the tests), and *Reg. exp. / tested site* (the number of regular expressions passed to each tested call site). The *mean* column describes the average value for each attribute. Columns *25%, 50%, 75%, 90%,* and *99%* show the distribution of each attribute at 25 percentile, median, 75 percentile, 90 percentile, and 99 percentile, respectively. The average number of tested regular expressions collected per project was 12 with a range of 1 to 2,004.

5.3.6 Regular Expression Characteristics. Table 3 shows the DFA information for regular expressions. *Nodes, edges, and edge pairs* are the total number of nodes, edges, edge pairs in the DFA graph of a regular expression. The average regular expression is quite large with 144 nodes, though this is skewed as the median is 28 nodes. *Regular exp. len.* measures the length of the string representing the regular expression itself in characters. *# Input strings* is the number of syntactically unique input strings executed by a project's test suite, per regular expression. The average number of syntactically unique test inputs per regular expression is 60, but the median is 2. *Input string len.* shows the lengths of the input strings (i.e., each $s \in S$) in terms of the number of characters.

5.4 Artifacts for RQ2

To explore the coverage of regular expressions using tools, we selected Rex [33] due to its high language feature coverage [9].

5.4.1 Artifact Selection. We need a set of regular expressions with the following characteristics: 1) are covered by tests; 2) can be analyzed by RE2 for coverage analysis; and 3) can be analyzed by Rex for test input generation. To satisfy 1) and 2), we begin with the dataset from RQ1 of 1,225 projects and 15,096 regular expressions. To satisfy 3), we select all the regular expressions that Rex supports and for which $|S_{succ}| > 0$, since Rex only generates matching strings, leaving 10,155 regular expressions of which 9,063 are syntactically unique.

5.4.2 Rex Setup. Rex defaults to *ManyMatch* as opposed to the *FullMatch* behavior of our dataset. To force Rex to treat each regular expression as a full match, we added endpoint anchors (i.e., ^ and $) to each regular expression. Because Rex may get stuck in generating input strings for certain regular expressions, we set a timeout of one hour for Rex to generate strings; regular expressions that exceed the timeout are discarded. Of the 10,155 regular expressions in GitHub whose $|S_{succ}| > 1$, Rex encountered the timeout for only two.

Table 4: Description of 7,926 regular expressions analyzed for RQ2. All numbers are rounded to nearest integer.

Attributes	mean	25%	50%	75%	90%	99%
Nodes ($\|N\|$)	220	13	31	162	618	970
Edges ($\|E\|$)	773	30	97	663	1,468	3,694
Edge pairs ($\|EP\|$)	2,422	36	186	1,021	1,999	21,274
Regular exp. len.	29	12	15	31	71	160
# Input strings ($\|S\|$)	70	1	2	8	39	961
$\|S_{succ}\|$	34	1	1	2	8	208

Another complication comes at the intersection of the Rex and RE2 language support; Rex-generated strings must be processed by RE2 for the coverage analysis. For example, the character class "\s" in Rex accepts six whitespace characters and RE2 accepts five. In another example, some generated Unicode strings in Rex could not be processed in RE2 because their Unicode encoding in Rex is UTF-16 while RE2 handles Unicode sequences encoded in UTF-8 or Latin-1. To simplify the experiment, we configured Rex to generated strings in ASCII. We also dropped strings which contain unsupported features or characters in either RE2 or Python 3. We also dropped strings which lead to failed matchings and reported the coverage based on successful matchings.

After filtering out all the unsupported regular expressions, our reported coverages by Rex strings in ASCII encoding are based on 7,926 regular expressions of 985 GitHub projects; 7,007 of them are syntactically unique. Table 4 shows the attributes of regular expressions for which Rex could generate strings.

5.4.3 Input String Generation. For each regular expression R, we use Rex to generate input string sets relative to the size of the matching strings $|S_{succ}|$. We generate input string sets of three sizes: equal to $|S_{succ}|$; equal to $5 \times |S_{succ}|$; and equal to $10 \times |S_{succ}|$. We refer to these experiments as *Rex1M*, *Rex5M*, and *Rex10M*, respectively. For each experiment, we repeated the string generation using the system time as the random seed to encourage diversity among the generated strings. The averages over five runs (*Rex5M* and *Rex10M*) or ten runs (*Rex1M*) for each metric are reported as Rex's coverage of R.

For example, say a regular expression R from GitHub has five input strings; $|S| = 5$. Three of the input strings are matching; $|S_{succ}| = 3$. For this experiment, Rex would generate three strings ten times, then 15 strings five times, then 30 strings five times, totaling $30 + 75 + 150 = 255$ generated strings. For each set of $\{3, 15, 30\}$ strings, NC, EC, and EPC are computed, averaged over $\{10, 5, 5\}$ runs.

In the case of finite languages, Rex may fail to generate sufficient input strings. For example, the total number of matching input strings in ASCII for a regular expression \d is ten (i.e., 0-9). If in the repository there are also three matching input strings, Rex could generate three strings ten times, but would fail to generate $5 \times 3 = 15$ strings. The calculation of NC, EC, and EPC are based on the best-effort: for each run of every regular expression, we calculate coverage with input strings up to $|S_{succ}|$ in *Rex1M*, 5x of $|S_{succ}|$ in *Rex5M*, and 10x of $|S_{succ}|$ in *Rex10M*; and coverage of every regular expression is the averages of its coverages over $\{10, 5, 5\}$ runs in *Rex1M*, *Rex5M*, and *Rex10M*. In other words, if

Rex failed to generate the required number of input strings, the coverage is calculated based on the input strings Rex can generate.

In the ten runs of generating input string sets equal to $|S_{succ}|$ for *Rex1M*, there are 833 regular expressions that have fewer input strings than $|S_{succ}|$ in at least one run. In the five runs of generating input string sets 5x of $|S_{succ}|$ for *Rex5M*, there are 2,041 regular expressions that have fewer input strings than 5x of $|S_{succ}|$ in at least one run. In the five runs of generating input string sets 10x of $|S_{succ}|$ for *Rex10M*, there are 2,336 regular expressions that have fewer input strings than 10x of $|S_{succ}|$ in at least one run.

6 RESULTS

Here, we present the results of RQ1 and RQ2 in turn.

6.1 RQ1: Test Coverage of Regular Expressions

We address RQ1 is two ways. First, we look at the number of call sites to FullMatch methods that are actually tested. Next, we look at the test coverage for each tested regular expression,

6.1.1 Tested Call Sites. In the 1,225 projects, there are 18,426 call sites of the instrumented functions in Section 5.1.1. However, only 3,093 call sites are executed by the test suites. This means that 15,333 (83.21%) of the call sites are not covered by the test suites. For those that are, the median of unique regular expressions per tested call site is one, with an average of five (Table 2).

Summary: Of the 18,426 call sites for FullMatch methods in 1,225 GitHub projects, only 3,093 (16.8%) are executed by the test suites.

6.1.2 Coverage of Tested Regular Expressions. We successfully generated static DFAs for 15,096 regular expressions from 1,225 Java GitHub projects and dynamic DFAs for 899,804 regular expression/input string pairs.[1] Among the regular expressions, 4,941 (32.7%) have only failing inputs (i.e., $|S_{succ}| = 0$) and 6,029 (39.9%) have only inputs of successful matching (i.e., $|S_{fail}| = 0$). This means that 10,970 (72.7%) of the regular expressions do not contain test inputs that exercise both the matching and non-matching scenarios. Of these, 6,318 (41.9%) regular expressions contain only one test string (i.e., $|S| = 1$). There are 4,126 (27.3%) regular expressions with both failed and successful matchings. [2]

Table 5 describes properties of the test input sets for each regular expression: $|S|$ is the size of the test suite, computed as the number of unique input strings for a regular expression; $|S_{succ}|$ means the number of matching inputs; $|S_{fail}|$ means the number of failing inputs; *succ_ratio* shows the ratio of successful matchings to all matchings for each regular expression; *fail_ratio* shows the ratio of failed matchings to all matchings for each regular expression. Generally, tested regular expressions use more failing inputs than successful inputs.

Table 6 describes the distributions of Node Coverage (NC), Edge Coverage (EC), and Edge-Pair Coverage (EPC) over S, S_{succ}, and S_{fail}. Figure 4 displays this information graphically, with coverage percentage on the y-axis and the input string sets, S, S_{succ}, and S_{fail} on the x-axis. Most of the regular expressions are not tested thoroughly since the mean values of coverage are low, especially for edge and edge-pair coverage. Although the coverages on failed matchings are relatively small, they contribute to a high overall test coverage. Failed matching tests are a necessary part of testing regular expressions, and as shown in Table 5, $|S_{fail}| > |S_{succ}|$.

Summary: A majority of regular expressions (10,970, 97.7%) are tested with exclusively passing (6,029, 39.9%) or exclusively failing (4,931, 32.7%) test inputs. Edge and edge-pair coverage are both very low. On average, the set of test inputs contains more failing inputs than successful inputs.

Table 5: Description of 15,096 Regular Expressions' test suites. All numbers are rounded to the nearest integer, except the ratios which are rounded to two decimal places.

Attributes	mean	25%	50%	75%	90%	99%
$\|S\|$	60	1	2	7	27	662
$\|S_{succ}\|$	19	0	1	1	4	79
$\|S_{fail}\|$	41	0	1	4	19	383
succ_ratio	49.03	0.00	44.70	100.00	100.00	100.00
fail_ratio	50.97	0.00	55.30	100.00	100.00	100.00

Table 6: Coverage values in Figure 4.

Coverage	Suite	mean	25%	50%	75%	90%	99%
NC (%)	S	59.05	24.62	63.64	95.65	100.00	100.00
NC (%)	S_{succ}	47.84	0.00	46.15	90.00	99.60	99.89
NC (%)	S_{fail}	18.89	0.00	8.51	25.00	62.26	100.00
EC (%)	S	28.74	6.67	23.90	49.97	53.80	80.00
EC (%)	S_{succ}	23.20	0.00	12.36	49.96	50.00	60.00
EC (%)	S_{fail}	8.55	0.00	2.20	7.80	32.19	65.08
EPC (%)	S	23.77	2.47	12.50	49.96	50.00	66.67
EPC (%)	S_{succ}	20.48	0.00	5.26	49.94	50.00	55.56
EPC (%)	S_{fail}	5.50	0.00	0.00	2.74	22.12	57.14

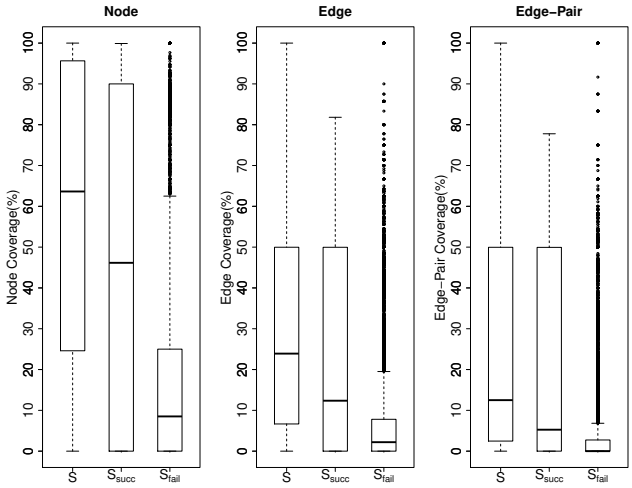


Figure 4: Coverage for 15,096 regular expressions.

6.2 RQ2: Coverage with Rex

Figure 5 shows the analysis results given the generated inputs in ASCII encoding, organized by each of five datasets. *Repo*$_B$*S* and *Repo*$_B$*M* show the coverages over S and S_{succ}, respectively, from 7,926 regular expressions using the developer-defined test suite in

[1] We note that 899,804 is less than $60 \times 15096 = 905760$ because the mean of *# Input strings* ($|S|$) is 59.60546 and rounded up to 60.

[2] Data at https://github.com/wangpeipei90/RegexTestingCoverageData.git.

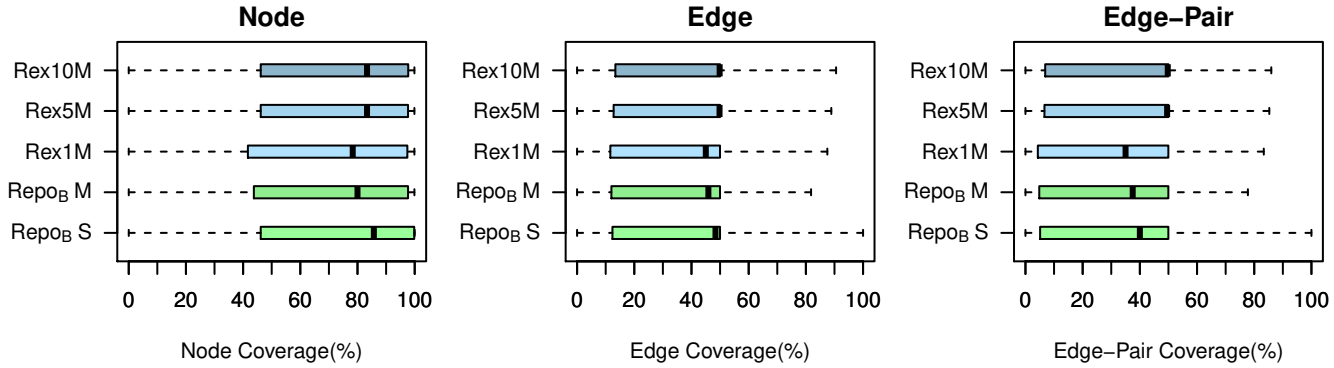


Figure 5: Node, edge, edge-pair coverage of 7,926 regular expressions with Rex-generated ASCII inputs (*Rex1M*, *Rex5M*, *Rex10M*) of 7,926 regular expressions in GitHub which are used in Rex (*Repo$_B$S*, *Repo$_B$M*).

Table 7: Coverage values of the 7,926 regular expressions in GitHub for *Repo$_B$M* and *Repo$_B$S* in Figure 5.

Coverage	Expr	mean	25%	50%	75%	90%	99%
NC (%)	*Repo$_B$M*	70.41	43.75	80.00	97.67	99.84	99.90
EC (%)	*Repo$_B$M*	33.79	12.01	45.91	49.97	50.00	66.67
EPC (%)	*Repo$_B$M*	29.39	4.83	37.50	49.97	50.00	60.00
NC (%)	*Repo$_B$S*	73.27	46.15	85.71	99.83	100.00	100.00
EC (%)	*Repo$_B$S*	36.35	12.36	48.39	49.97	60.00	85.71
EPC (%)	*Repo$_B$S*	30.68	5.13	40.00	49.97	50.00	74.67

Table 8: Coverage values of the 7,926 regular expressions using Rex for *Rex1M*, *Rex5M*, and *Rex10M* in Figure 5.

Coverage	Expr	mean	25%	50%	75%	90%	99%
NC (%)	*Rex1M*	69.29	41.67	78.33	97.44	99.84	99.90
EC (%)	*Rex1M*	33.57	11.62	45.00	49.97	50.00	71.43
EPC (%)	*Rex1M*	29.50	4.33	35.00	49.96	50.00	66.67
NC (%)	*Rex5M*	71.69	46.15	83.33	97.67	99.84	99.90
EC (%)	*Rex5M*	36.42	12.77	49.81	50.00	54.55	80.00
EPC (%)	*Rex5M*	33.04	6.63	49.54	50.00	56.67	75.00
NC (%)	*Rex10M*	72.01	46.15	83.33	97.73	99.84	99.90
EC (%)	*Rex10M*	36.87	13.39	49.85	50.00	55.89	80.00
EPC (%)	*Rex10M*	33.77	6.90	49.77	50.00	58.33	75.00

GitHub; details are in Table 7. *Rex1M*, *Rex5M*, and *Rex10M* show the coverages of 7,926 regular expressions based on the Rex-generated test inputs with sizes of 1x, 5x, and 10x of $|S_{succ}|$ the user-defined test suite, respectively. Coverage details are shown in Table 8.

Table 9 illustrates the differences in coverage between the repository (*Repo$_B$M* and *Repo$_B$S*) and Rex (*Rex1M*, *Rex5M*, and *Rex10M*). Using a paired Wilcoxon signed-rank test, we find that for all three coverage metrics, *Repo$_B$M* significantly outperforms *Rex1M* with $\alpha = 0.0001$. However, as test suite size is strongly correlated with coverage [19], as soon as the Rex test set is amplified to 5x and 10x the size, the coverage of Rex outperforms the developer coverage. When considering all test inputs from the repository and not just the successful ones, with test inputs sets of the same size, *Repo$_B$S* outperforms *Rex1M*. However, this comparison is unfair since Rex does not generate non-matching strings. That said, as soon as the Rex dataset is amplified as in *Rex5M* and *Rex10M*, there is no clear winner compared to all test inputs from the repository. While it may appear that Rex can do as well as the repository, the reality is that the error node will never be covered by Rex, a fact which is not apparent by looking at the numbers alone.

Table 9: Differences in coverage based on datasets in Figure 5. Hypothesis tests used paired Wilcoxon signed-rank test. Bold text identifies when one of the datasets had significantly higher coverage for all three metrics. If there was a conflict between the metrics (e.g., Set1 > Set2 for NC, and Set1 < Set2 for EPC), there was no winner

		$H_0 : Set1 \stackrel{d}{=} Set2$		
Set1	Set2	NC	EC	EPC
Repo$_B$M	*Rex1M*	$p < 0.0001$	$p < 0.0001$	$p < 0.0001$
Repo$_B$M	***Rex5M***	$p < 0.0001$	$p < 0.0001$	$p < 0.0001$
Repo$_B$M	***Rex10M***	$p < 0.0001$	$p < 0.0001$	$p < 0.0001$
Repo$_B$S	*Rex1M*	$p < 0.0001$	$p < 0.0001$	$p < 0.0001$
Repo$_B$S	*Rex5M*	$p < 0.0001$	$p = 0.0004$	$p < 0.0001$
Repo$_B$S	*Rex10M*	$p < 0.0001$	$p = 0.4147$	$p < 0.0001$
Repo$_B$S	*Repo$_B$M*	$p < 0.0001$	$p < 0.0001$	$p < 0.0001$

Summary: Rex can handle approximately 78.1% of the regular expressions from our dataset. Considering only the matching test inputs and test sets of the same size, Rex does not achieve coverage as high as the developer-written tests. However, the coverage numbers are extremely close. This indicates that tools such as Rex can be used to write test inputs with similar coverage to the developer tests, but will always miss N_e and all edges incident to it.

7 DISCUSSION

This section summarizes future work based on our findings and discusses threats to validity.

7.1 Opportunities For Future Work

Coverage provides useful stopping criteria for testing. However, high coverage does not necessarily imply test suite effectiveness in source code [19], which may also hold true for regular expressions. At the same time, as regular expressions are responsible for many software faults, it is important to explore how to make them less error-prone. Our approach in this work is through test metrics, and there are many areas of future work that follow:

String-generation tools: Given the low coverage of regular expressions shown in Figure 4, a natural next step could be to generate strings to achieve high coverage. Adding a mutation step to the input string may be effective at forcing the Rex-generated strings into the error state to cover the uncovered edges and node. An alternate approach may be to provide the complement of the regular expression to Rex as another way to generate failing inputs.

With automatically-generated strings, one threat is usability. For the developer-written tests, it is likely that the regular expression strings are more meaningful in context than they are for the Rex-generated strings. Future work will look at the overlap in content between the test inputs from the repository and from Rex.

However, it may not always be possible to achieve 100% test coverage, even with a perfect string generation tool. There are regular expressions that are untested because they are unreachable. Some regular expressions have hard-coded matching inputs, which makes it impossible to improve the coverage; for example: `boolean isMatch = Pattern.matches("a*b", "ab");` Future work for improving coverage levels should also consider the potential for improvement based on such factors.

Beyond Structural Coverage: The metrics we explore are structural metrics, which can identify faults that are revealed in the structure of the DFA, such as the example in Section 2.3. Alternately, as suggested in prior work [10], refactoring could potentially reveal this particular fault, as the numeric representation `[0-9]` was found to be more understandable than `\d`. Performing the replacement might alert the developer that `d` should be `\d`.

In terms of improving regular expression testing, structural metrics are a first step. Building on the example in Section 3, achieving 100% coverage requires a minimum number of test inputs that vary in string length and content. In the example of `\d+`, there are strings of length one to length four, though strings could be longer to test multiple iterations on the self-loop. Strings can contain only digits, only non-digits, or both digits and non-digits. Strings can start with digits or start with non-digits. Defining such input space partitions may lead to intuitive test sets with high behavioral coverage.

7.2 Threats to Validity

Internal: We measure the test coverage of regular expression used in functions of full matching with FullMatch DFAs in the forward direction. The experimental results may not reflect the test coverage of regular expressions used in other functions, nor the test coverage of regular expressions which could not be converted into a DFA.

External: The Java regular expressions used in this evaluation were collected from RepoReaper Java Maven projects compiled with Java jdk1.7, which is only a small portion of all GitHub Java projects and may not generalize to all Java projects and to other languages. It is possible that there are still regular expressions from third-party libraries in the dataset, which could bias results. Due to limitations of RE2 and Rex, the results of test coverage applies exclusively to the features supported. All our projects had test suites, which may overestimate the test coverage levels for typical regular expressions.

8 RELATED WORK

Regular expressions are used widely in software programs [9] but are often difficult to understand and error-prone [10]. Prior work on regular expression comprehension [10] raises a concern about how well the regular expressions used in programs are tested. Although there are papers on program test coverage, none of them have specifically discussed testing regular expressions.

Software test coverage can be measured at different levels of granularity, such as method, statement, branch, integration, and unit (e.g., [2, 24, 27, 31, 37]). Symbolic execution [3, 7, 8, 35] is one way to generate inputs and to obtain program test coverage at the level of branches. There are many tools for automated test generation [16, 30, 36]. For example, Reggae [25] aims to mitigate the large space exploration issues in generating test inputs for programs with regular expressions.

With respect to the finite automaton constructed from regular expressions, `brics` [28] contains a DFA implementation with very limited operations; while RE2 [13, 14] provides a DFA implementation which runs much faster than traditional regular expression engines. Rex [33] builds a symbolic representation of finite automata (SFA). Some string solvers [21] and tools for generating testing inputs which use string solvers [18, 34] build finite-state automata based on string constraints.

Visualizations to aid debugging [1, 6] are powerful techniques for regular expression comprehension, and may provide some explanation for low test coverage of regular expressions in source code, that is, developers use online tools instead.

Other techniques and tools have been developed in string generation or regular expression extraction for system fault detection and performance optimization. Rex [33] generates testing inputs for the regular expression according to its SFA representation. `brics` [28] generates inputs by traversing the DFA and building strings from the smallest bytes to the largest bytes of every DFA states. Some string generation tools need user-specified string length [18, 21, 28]. EGRET [23] is focused on generating unexpected test strings to expose the regular expression errors, but it is based on common mistakes when creating regular expression rather than maximizing test coverage of regular expressions. MUTREX [4] employs distinguishing strings which can separate a mutated regular expression from the original one to expose system faults. Genetic programming has also been applied [12] to find equivalent alternative regular expressions which exhibit improved performances.

9 CONCLUSION

In this paper we explore coverage over the DFA representation of a regular expression and measure coverage of regular expressions from 1,225 GitHub Java Maven projects. We find that over 80% of *FullMatch* functions are not tested and that most of the tested regular expressions have a low edge and edge-pair coverage. We also show that with the help of the regular expression tool Rex it is possible to improve the regular expression testing coverage by adding input strings, but that there is an upper bound for this type of improvement. This work is a first step toward better understanding how regular expressions are tested in the wild; future work will explore how various coverage metrics can reduce the bugs associated with regular expressions.

ACKNOWLEDGMENTS

This work is supported in part by NSF-SHF #1714699 and #1645136.

REFERENCES

[1] [n. d.]. Online regex tester, debugger with highlighting for PHP, PCRE, Python, Golang and JavaScript. https://regex101.com/.

[2] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing.* Cambridge University Press.

[3] Saswat Anand, Corina Păsăreanu, and Willem Visser. 2007. JPF–SE: A symbolic execution extension to Java pathfinder. *Tools and Algorithms for the Construction and Analysis of Systems* (2007), 134–138.

[4] P. Arcaini, A. Gargantini, and E. Riccobene. 2017. MutRex: A Mutation-Based Generator of Fault Detecting Strings for Regular Expressions. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 87–96. https://doi.org/10.1109/ICSTW.2017.23

[5] Rohit Babbar and Nidhi Singh. 2010. Clustering Based Approach to Learning Regular Expressions over Large Alphabet for Noisy Unstructured Text. In *Proceedings of the Fourth Workshop on Analytics for Noisy Unstructured Text Data (AND '10)*. ACM, New York, NY, USA, 43–50. https://doi.org/10.1145/1871840.1871848

[6] Fabian Beck, Stefan Gulan, Benjamin Biegel, Sebastian Baltes, and Daniel Weiskopf. 2014. RegViz: visual debugging of regular expressions. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 504–507.

[7] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems*. ACM, 183–198.

[8] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[9] Carl Chapman and Kathryn T Stolee. 2016. Exploring regular expression usage and context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 282–293.

[10] Carl Chapman, Peipei Wang, and Kathryn T Stolee. 2017. Exploring regular expression comprehension. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 405–416.

[11] Shigeru Chiba. 1998. Javassist-a reflection-based programming wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vol. 174. 21.

[12] Brendan Cody-Kenny, Michael Fenton, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. 2017. A search for improved performance in regular expressions. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 1280–1287.

[13] Russ Cox. 2007. Regular expression matching can be simple and fast (but is slow in Java, Perl, PHP, Python, Ruby,...). *URL:http//swtch.com/~rsc/regexp/regexp1.html* (2007).

[14] Russ Cox. 2010. Regular expression matching in the wild. *URL:http//swtch.com/~rsc/regexp/regexp3.html* (2010).

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[16] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2013. Does automated white-box test generation really help software testers?. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 291–301.

[17] Jeffrey EF Friedl. 2002. *Mastering regular expressions*. " O'Reilly Media, Inc.".

[18] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. 2013. JST: An Automatic Test Generation Tool for Industrial Java Applications with Strings. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 992–1001. http://dl.acm.org/citation.cfm?id=2486788.2486925

[19] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 435–445. https://doi.org/10.1145/2568225.2568271

[20] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2013. HAMPI: A Solver for Word Equations over Strings, Regular Expressions, and Context-free Grammars. *ACM Trans. Softw. Eng. Methodol.* 21, 4, Article 25 (Feb. 2013), 28 pages. https://doi.org/10.1145/2377656.2377662

[21] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. 2009. HAMPI: a solver for string constraints. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 105–116.

[22] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[23] Eric Larson and Anna Kirk. 2016. Generating Evil Test Strings for Regular Expressions. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*. IEEE, 309–319.

[24] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. 2009. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*. IEEE, 220–229.

[25] Nuo Li, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Reggae: Automated test generation for programs using complex regular expressions. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*. IEEE, 515–519.

[26] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and H. V. Jagadish. 2008. Regular Expression Learning for Information Extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP '08)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 21–30. http://dl.acm.org/citation.cfm?id=1613715.1613719

[27] Yashwant K Malaiya, Michael Naixin Li, James M Bieman, and Rick Karcich. 2002. Software reliability growth with test coverage. *IEEE Transactions on Reliability* 51, 4 (2002), 420–426.

[28] Anders Møller. 2017. dk.brics.automaton – Finite-State Automata and Regular Expressions for Java. `http://www.brics.dk/automaton/`.

[29] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253.

[30] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 815–816.

[31] Paul Piwowarski, Mitsuru Ohba, and Joe Caruso. 1993. Coverage measurement experience during function test. In *Proceedings of the 15th international conference on Software Engineering*. IEEE Computer Society Press, 287–301.

[32] Michael Sipser. 2006. *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology Boston.

[33] Margus Veanes, Peli De Halleux, and Nikolai Tillmann. 2010. Rex: Symbolic regular expression explorer. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE, 498–507.

[34] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. 2008. Dynamic test input generation for web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 249–260.

[35] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. 2005. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution.. In *TACAS*, Vol. 3440. Springer, 365–381.

[36] Sai Zhang, David Saff, Yingyi Bu, and Michael D Ernst. 2011. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 353–363.

[37] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software unit test coverage and adequacy. *Acm computing surveys (csur)* 29, 4 (1997), 366–427.